IEEE Symposium on Technologies for Homeland Security 2016

Programming Language Support for Analyzing Non-Persistent Data

Yung-Hsiang Lu, Milind Kulkarni, Nouraldin Jaber
School of Electrical and Computer Engineering
Purdue University
West Lafayette, Indiana 47907-2035, USA
yunglu@purdue.edu, milind@purdue.edu, njaber@purdue.edu

Jerry Xiaojin Zhu
Department of Computer Sciences
University of Wisconsin
Madison, Wisconsin, 53706-1613, USA
jerryzhu@cs.wisc.edu

***Abstract*—For safety and security, surveillance cameras are widely deployed. A high percentage of the visual data, however, is never watched by humans nor analyzed by computer programs. Moreover, it is common practice to erase the data after a short duration (say, two weeks) and reuse the storage space. As a result, the data are *non-persistent*. Non-persistent data presents serious security risks: the unwatched and unanalyzed data may include evidence of security breaches. After the data is erased, it is no longer possible to detect the breaches nor prosecute the suspects. This paper proposes a potential solution to remedy this situation by adding *automatic* data sampling to a programming language. If a piece of data is marked as non-persistent, the compiler and the run-time system automatically sample and store the data, hence making a small fraction of the data persistent. The samples would allow post-event analysis to detect security breaches that are not detected earlier. The samples, due to the much smaller sizes compared with the original non-persistent data, may be analyzed using more sophisticated computer programs that are unable to keep up with the speeds of data generation.**

I. Introduction

Surveillance cameras (commonly called CCTV or close-circuit television) are widely deployed for safety and security. The visual data may be managed in one of the following ways: (1) For areas of high levels of security, dedicated personnel watch the visual data attentively in real-time. (2) The data is checked occasionally by people that have other job functions (such as receptionists). (3) The data is saved and watched by humans only if incidents (through other channels) are reported. (4) The data is not watched and not saved.

In many cases, the data is recorded and kept for short durations (such as two weeks) and the storage media are reused if no event is reported within this duration. The data is considered *non-persistent data* (NPD) because it is available for only short durations. Many other reasons can also make data non-persistent [1]; for example, analyzing the data may take too long and timely decisions must be made. Streaming data is more likely non-persistent if the data is not analyzed immediately while it is generated. Non-persistent data can have profound implications for security. If the data is not watched by humans or analyzed by computer programs, security breaches are not detected. If the data is erased, detecting security breaches and prosecuting suspects would be impossible. As the amounts of data (especially visual data) grow rapidly, it is expected that non-persistent data will be an increasingly serious problem. One report estimates that only 0.5% of data is ever analyzed [2].

Various solutions can remedy this situation. Some obvious solutions are (i) keeping the data and never erasing it; (ii) hiring enough staff to watch the data; (iii) adopting sophisticated computer programs to analyze the data in real-time. These solutions are too costly and impractical. Another solution relies on additional channels to report security breaches and then retrieves the recorded data to confirm the breaches and to serve as evidence. This is the commonly adopted solution. This solution has a major drawback, however: it assumes the existence and validity of the additional channels.

		Detection	
		Yes	No
Intruder	Yes	True Positive	False Negative
	No	False Positive	True Negative

Fig. 1. Four possible scenarios of intruder and detection.

Consider a surveillance camera monitoring a restricted area and detecting unauthorized passage (i.e., an intruder). The data is analyzed by a computer program. There are four possible scenarios as shown in Figure 1.

1) *True Positive*: there is an intruder and the computer program detects the intruder.
2) *False Positive*: There is no intruder and the computer program detects an intruder.
3) *False Negative*: There is an intruder but the computer program fails to detect the intruder.
4) *True Negative*: there is no intruder and the computer program does not detect any intruder.

When the computer program detects an intruder, the program can store the data and make the data persistent. The main problem is that it is *impossible to discover false negatives*. There is no second chance to correct the mistake. This paper proposes a new option: creating a programming language and the accompanying run-time system that automatically sample a small portion of the non-persistent data and store the samples in non-volatile media. As a result, the samples become persistent and can be used to (probabilistically) discover

false negatives. This paper extends our previous work [1] by providing a conceptual design and use cases.

II. Related Work

The prior work can be divided into several categories: data processing languages, languages that operate over sampled data, languages that operate over uncertain or approximate data, and compressive sensing.

MapReduce [3] and Dryad [4] allow programmers to develop analysis pipelines consisting of multiple stages, each of which applies a different type of processing to data. MapReduce focuses on applications with two stages of processing: a *map* stage that applies a function to each element of an input data source, and a *reduce* stage that aggregates the results of the map stage. Dryad allows programmers to build more complex operator pipelines, in a dataflow-like manner. Both require programmers to write operations for each stage using low-level languages such as Java. To ease the challenge of writing programs in systems like MapReduce and Dryad, higher level programming languages are introduced to allow applications to express using high level, often SQL-like constructs which are ultimately compiled down into MapReduce or Dryad pipelines, or similar systems. Examples of such languages include Pig Latin [5], DryadLINQ [6], Sawzall [7] and GLADE [8]. Most of them focused on *batch processing* programs and not designed to process data streams (such as surveillance video). Languages like Hadoop-online [9], SPC [10] and Apache Storm [11] used MapReduce-like strategies to tackle streaming data. Even though these languages raised the level of abstraction, making it easier for domain experts to write analysis programs, none of them is designed to handle NPD.

Aqua [12], STRAT [13], and BlinkDB [14] support *approximate queries*, where an SQL-like query can be answered approximately, with some confidence. These languages target a restricted class of problems (database queries), and are not suited for more complex data analysis tasks. Moreover, they often rely on preprocessing a large, existing database; they are not designed to handle NPD or provide support for streaming or validation.

Hazy [15–17] is a data processing system, integrated with statistical processing methods. Many database systems have also developed approaches to handling uncertain data [18–20]. Even though these system consider uncertainty in data, they do not report confidence level about false negative.

Compressive sensing [21, 22] is a data summarization technique, assuming that the data stream has a sparse representation under some basis. Compressive sensing takes random measurements, in the form of a sensing matrix. Compressive sensing is not designed to handle NPD because a random measurement has to be artificially applied to the data stream. Moreover, compressive sensing must find a basis in which the data stream has a sparse representation; this is generally difficult.

III. Sampling Strategies

A desirable sampling strategy should have the following properties: (1) Sampling should be lightweight, adding only negligible load to the entire system. This is essential because a restricted area may have many surveillance cameras and heavy computation would be undesirable. (2) It should save only a small amount of data. (3) It should have a high chance of detecting false negative.

An obvious sampling strategy is *regular downsampling*. If the sampling period is n, this method saves one sample after skipping $n-1$ pieces of data. This method can be easily tuned by adjusting the value of n. Regular downsampling has the advantage of simplicity but suffers from the aliasing problem: If an event occurs periodically and the period happens to be n, there is a large $\frac{n-1}{n}$ probability that this event is never detected. Aliasing also occurs when the event's period is a multiple of n.

One improvement over regular downsampling is *uniform sampling*. Instead of skipping exactly $n-1$ pieces of data, uniform sampling chooses an integer number x between 0 and $2(n-1)$ uniformly in every period. The method skips x pieces of data and saves the next item. By varying x, this method is able to avoid the aliasing problem mentioned earlier.

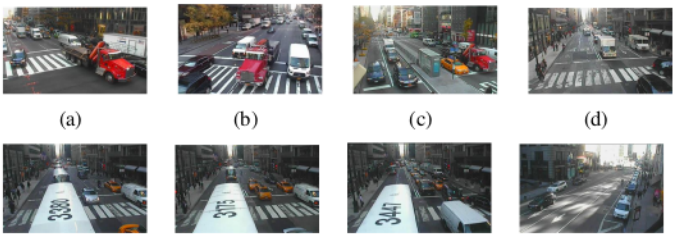

Fig. 2. (a)-(c) Detect a red truck at three different intersections. (d) No red object is detected. (e)-(g) Recognize the bus numbers. (h) No bus is detected. Source: `http://nyctmc.org/`.

The two methods mentioned above assume that every piece of data has the same importance. This is usually false, in particular in surveillance video. Surveillance video can usually be filtered by simple preprocessing to determine the likelihood of *importance*. Consider the two sets of examples in Figure 2. In the first set shown in Figure 2 (a)-(d), the goal is to detect the movement of a red truck and a simple program can first determine whether any red object appears in the video streams. If no red object appears, such as Figure 2 (d), this frame can be sampled with very low probability (discarded with high confidence). Conversely, after detecting a red object the frame will be sampled with a higher probability. For Figure 2 (e)-(g), the goal is to determine the numbers on the bus roofs. If no black numbers on a white vehicle top is detected, such as Figure 2 (h), this frame can similarly be sampled with very low probability.

The concept of *importance sampling* is illustrated in Figure 3. The data streams from many video cameras are analyzed on-site (at the cameras) using image processing programs. These computer programs can be fairly simple but may substantially reduce the data rates. In this figure, the importance of each stream is encoded by the darkness of the arrows: a more important stream is expressed as a darker arrow. After marked

by the importance, the data streams are sent to a sampling program that determines which parts of the data are to be saved in persistent storage.

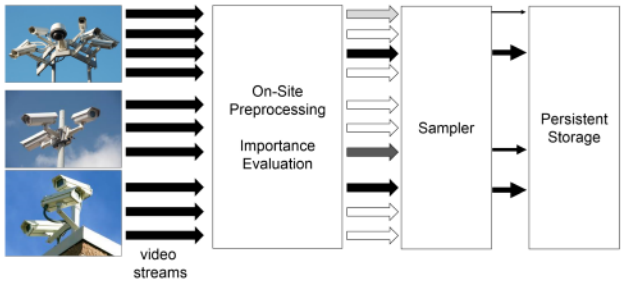

Fig. 3. Importance sampling. Simple preprocessing is used to determine the likelihood of importance of each data stream. Then the sampler determines whether any sample should be saved in persistent storage.

Compared with regular downsampling or uniform sampling, importance sampling has a greater chance of detecting false negatives while saving smaller amounts of data. The importance evaluators are simple and lightweight programs and can be deployed directly at the data sources (i.e., cameras). As the importance factor may be different, the evaluators must be reprogrammable through networks.

IV. Conceptual Design

This section sketches a design of a system that incorporates the sampling strategies outlined above to help write applications that analyze non-persistent data.

At a high level, we can think of programs that analyze non-persistent data as *streaming* programs: programs that process incoming streams of data (in this case, NPD). Most streaming systems focus on data analysis tasks with fixed behaviors (for example, MapReduce-like programs, where a single analysis task is performed over the incoming data). However, for use cases such as analyzing incoming camera data, the functionality of such a streaming program can change in response to events. For example, when an intruder is detected, the data from the camera that detected the intruder should now be processed to perform object tracking, rather than object detection; the functionality of the system changes.

To accommodate this usage scenario, we envision programs written in our system to be a set of communicating *agents* (similar to the actor model [23]). Hence, each camera is associated with an agent, and there might be other computational agents as well. Each of these agents can communicate with one another, as well as execute tasks on their incoming data streams. Crucially, each of these agents can also *schedule new behaviors* for other agents (in other words, in addition to directing data to other agents, an agent can also direct computation to other agents). In this way, for example, when a camera detects an intruder, it can not only change its own computational configuration to begin performing object tracking, it can also direct agents associated with other cameras to also perform object tracking, allowing the intruder to be tracked across the network of cameras once it is detected.

This system architecture can be integrated with sampling: taking a sample of input data is merely another computational action that an agent can perform. Because computational configurations can change, this design allows sampling frequencies and types to change in response to events, and even for agents to direct other agents to change their sampling behaviors. Hence, for example, once an intruder is detected, the camera that detected the intruder can direct nearby agents to sample more frequently, to ensure that the moment that the intruder crosses into another camera's field of view is saved.

V. Conclusion

The problem of non-persistent data plagues security settings, where the data that provide evidence of security concerns may be discarded, either because automatic systems do not identify the importance, or because of the costs of storing data. Mitigating this problem requires carefully saving small portions non-persistent data, choosing the data to be preserved for longer periods of time. Doing so requires consideration of many issues regarding importance, rates of preservation, etc. Different surveillance systems could have different factors deciding importance, make any manual process undesirable and error-prone. The proposed solution is well-designed programming language supports that can aid in this process: programmers can express applications that analyze incoming data, and a combination of compilers and runtime systems can automatically determine which portions of the data to preserve, both to retain signatures of events and to provide some confidence that false negatives are avoided.

Acknowledgment

The authors would like to thank Professor Tiark Rompf at Purdue University for the valuable suggestions.

References

[1] Milind Kulkarni and Yung-Hsiang Lu. Beyond big data-rethinking programming languages for non-persistent data. In *International Conference on Cloud Computing and Big Data*, 2015.

[2] Manju Bansal. Big data: Creating the power to move heaven and earth, mit technology review. http://www.technologyreview.com/view/530371/big-data-creating-the-power-to-move-heaven-and-earth/, September 2 2014.

[3] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Conference on Symposium on Opearting Systems Design & Implementation*, pages 10–10, 2004.

[4] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 59–72, 2007.

[5] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: A not-so-foreign language for data processing. In *ACM SIGMOD*

International Conference on Management of Data, pages 1099–1110, 2008.

[6] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *USENIX Conference on Operating Systems Design and Implementation*, pages 1–14, 2008.

[7] Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. Interpreting the data: Parallel analysis with sawzall. *Sci. Program.*, 13(4):277–298, October 2005.

[8] Yu Cheng, Chengjie Qin, and Florin Rusu. Glade: Big data analytics made easy. In *ACM SIGMOD International Conference on Management of Data*, pages 697–700, 2012.

[9] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, John Gerth, Justin Talbot, Khaled Elmeleegy, and Russell Sears. Online aggregation and continuous query support in mapreduce. In *ACM SIGMOD International Conference on Management of Data*, pages 1115–1118, 2010.

[10] Lisa Amini, Henrique Andrade, Ranjita Bhagwan, Frank Eskesen, Richard King, Philippe Selo, Yoonho Park, and Chitra Venkatramani. Spc: A distributed, scalable platform for data mining. In *International Workshop on Data Mining Standards, Services and Platforms*, pages 27–37, 2006.

[11] Apache storm project. http://storm-project.net.

[12] Swarup Acharya, Phillip B. Gibbons, Viswanath Poosala, and Sridhar Ramaswamy. The aqua approximate query answering system. In *ACM SIGMOD International Conference on Management of Data*, pages 574–576, 1999.

[13] Surajit Chaudhuri, Gautam Das, and Vivek Narasayya. Optimized stratified sampling for approximate query processing. *ACM Trans. Database Syst.*, 32(2), June 2007.

[14] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. Blinkdb: Queries with bounded errors and bounded response times on very large data. In *ACM European Conference on Computer Systems*, pages 29–42, 2013.

[15] Arun Kumar, Feng Niu, and Christopher Ré. Hazy: Making it easier to build and maintain big-data analytics. *Queue*, 11(1):30:30–30:46, January 2013.

[16] Feng Niu, Ce Zhang, Christopher Ré, and Jude Shavlik. Elementary: Large-scale knowledge-base construction via machine learning and statistical inference. *IJSWIS Special Issue on Web-Scale Knowledge Extraction*, 2012.

[17] Feng Niu, Ce Zhang, Christopher Ré, and Jude Shavlik. Deepdive: Web-scale knowledge-base construction using statistical learning and inference. *Second Int.l Workshop on Searching and Integrating New Web Data Sources*, 2012.

[18] D. Barbará, H. Garcia-Molina, and D. Porter. The management of probabilistic data. *IEEE Trans. on Knowl. and Data Eng.*, 4(5):487–502, October 1992.

[19] Omar Benjelloun, Anish Das Sarma, Alon Halevy, and Jennifer Widom. Uldbs: Databases with uncertainty and lineage. In *Proceedings of the 32Nd International Conference on Very Large Data Bases*, VLDB '06, pages 953–964. VLDB Endowment, 2006.

[20] Nilesh Dalvi and Dan Suciu. Management of probabilistic data: Foundations and challenges. In *ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 1–12, 2007.

[21] Richard Baraniuk. Compressive sensing. *IEEE signal processing magazine*, 24(4), 2007.

[22] David L Donoho. Compressed sensing. *Information Theory, IEEE Transactions on*, 52(4):1289–1306, 2006.

[23] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986.